Amendments to the Specification:

Please replace paragraph 2 on page 1 with the following amended paragraph:

CROSS-REFERENCES TO RELATED APPLICATIONS

This Continuation-in-part application is related to co-pending U.S. Patent Application No. ~~—/———~~ 09/838,550 (Attorney Docket 20181-51), filed April 18, 2001, entitled "Method For Effective Binary Translation Between Different Instruction Sets Using Emulated Supervisor Flag And Multiple Page Tables;" U.S. Patent Application No. ~~—/———~~ 09/838,532 (Attorney Docket 20181-50), filed April 18, 2001, entitled "Method for Fast Execution of Translated Binary Code Utilizing Database Cache for Low-Level Code Correspondence;" and U.S. Patent Application No. ~~—/———~~ 09/838,530 (Attorney Docket 20181-55), filed April 18, 2001, entitled "Method for Emulating Hardware Features of a Foreign Architecture in a Host Operating System Environment" each of which is incorporated herein by reference as if set forth in full in this document.

Please replace paragraph 2 beginning on line 8 on page 4 with the following amended paragraph:

Different approaches to maintaining precise exceptions are known in the art. For example, some systems may create a "check point" in the code at which point the normal operation of the system is suspended to save the current state. However, suspending the execution is inherently undesirable if the system is to operate without degrading performance compared to the foreign architecture. In alternative approaches, some systems may employ "speculative execution" where all branches of a code sequence are executed and stored for later use. However, this approach ~~is~~ results in an inefficient utilization of system resources.

Please replace paragraph 2 beginning at line 31 on page 7 with the following amended paragraph:

Referring now to Figure 1, a host computer system 100 based on explicit parallelism and wide instruction words architecture with hardware assistance for efficient binary translation is illustrated. Host computer system 100 comprises a computer processing unit (CPU) 102 and a memory management unit (MMU) 104. Host CPU 102 comprises one or more execution units 106 and a call/return cache 108. Execution units 106 include logic to input and retrieve address pairs from cache 108 to facilitate the execution of binary translated code. When translating foreign code, execution units 106 input a foreign address to the call/return cache and retrieve a corresponding host address for use during execution of binary translated code. The CPU 102 also includes a register file 110. Under software control, the registers are explicitly renamed at compile time for holding temporary data or foreign data. The same foreign register may be located in various host registers at run-time. MMU 104 includes logic to form a foreign virtual memory space 116 and logic to form a host virtual memory space 118. The MMU 104 also includes a translation lookaside buffer (TLB) 120 designed to provide translation from virtual to physical addresses and to provide coherence between foreign code in foreign virtual memory 116 and binary translated code in host virtual memory 118. Any access ~~of~~ to foreign memory is handled by TLB 120. The dual virtual memory spaces 116 and 118 are used and maintain the content of the foreign virtual memory in a consistent state with the foreign architecture during foreign code execution.

Please replace paragraph 3 beginning on line 29 on page 6 and ending on page 7 line 9 with the following amended paragraph:

In accordance with another embodiment of the present invention, a computer system employs a register file for storing temporary values and foreign registers rather than fixed registers in optimized binary translated code for preserving x86's general-purpose registers. Rather than use multiple register sets where one register set shadows the foreign architecture and another set as working registers, the computer system uses a unified register file. The register file is organized with an overlapping window with explicit register renaming of the foreign system's registers. The explicit renaming enables the optimizing translation processes (compilation) to aggressively

optimize foreign code. During compilation, state information is saved at a plurality of recovery points in the binary translated code. The saved information includes information that describes which registers correspond to the general purpose registers of the foreign architecture and this information is saved in external memory, e.g. on hard disk or in flash memory. Accordingly, there is no explicit hardware correspondence between the foreign register set and the host registers. The exception handler in the host architecture maintains documentation showing which registers must be used to restore the original foreign register content.

Please replace paragraph 3 beginning at line 32 on page 8 with the following amended paragraph:

FIG. 2 shows a block diagram of computer system 100 comprising host CPU 102 and MMU 104 together with hardware support for efficient and reliable execution of binary translated code. Through a software layer ~~208~~ 206, CPU 102 operates to execute foreign code on the host system. The process of decoding and semantic substitution is fulfilled in binary translator software. Foreign code is maintained in foreign virtual space 116 while the host processes are maintained in host virtual space 118. After semantic substitution for foreign operations in terms of host operations, the intermediate representation is processed by an optimizing binary translation process 202 to improve performance. Binary translation process 202 does not change the sequence of memory write instructions or, to be more specific, the sequence in which memory write operations (store operations) are performed is the same as in the foreign code. Accordingly, memory write side effects coincide with the behavior expected on a platform based on the foreign architecture. But to optimize performance, load operations can be moved ahead of store operations and to avoid address conflicts, there is a dedicated hardware buffer for address comparing (disambiguation memory). Maintaining the correct order of side effects for load operations is achieved with the present invention as described more fully below.

Replace paragraph 2 beginning at line 6 on page 15 with the following amended paragraph:

By way of example, for the x86's code sequence:

ADD ~~ebx~~ edx, [edi + 0x10]

```
    ADD ecx, esi
    MOV [edi + 0x8], ecx
    CMP [edi + 0xc], esi
    JNC Label
    SUB edx, [edi + 0x14]
Label:
    MOV [ecx], edx
    SRL esi, 1
```

Replace paragraph 1 beginning at line 10 on page 16 with the following amended paragraph:

The foreign code fragment shown above begins by performing an ADD operation. This operation first obtains the value at a memory location identified by summing the contents of the EDI register and a constant, 0x10. The value from memory is then added to the contents of the ~~EBX~~ EDX register. This operation is emulated by the first two RISC-like instructions depicted on the left. Specifically, in the host code, a load operation is executed to move the value stored in memory to a register R1. Then, an ADD,f operation adds the contents of register R1 to the contents in the EDX register with the result stored in the EDX register and obtains condition codes in the host register FLAGS in the host register file.

Replace paragraph 1 beginning at line 1 on page 18 with the following amended paragraph:

2 SetIP; ADD r1, edx, r1; ST r2, [edi+0x8]; CMPnc r4, esi -> p[0]; CMP,f r4,esi, ~~r3~~r6
3 SRL,f esi, 1, r4 & r5; SUB,f r1, r5, r1 & ~~r3~~ r6 (~p[0])
4 SetIP; ST r1, [r2]

Replace paragraph 1 beginning at line 1 on page 18 with the following amended paragraph:

The second point is described by ~~the same~~this documentation because registers R1, R2 and R3 have been released by the host optimizing scheduler and then reused in further calculations. After finishing execution of the code the documentation will have the following contents:

Replace paragraph 4 beginning at line 24 on page 15 with the following amended paragraph:

```
1   LD [edi + 0x10], r1                #
2   ADD,f r1, edx, edx @ & flags       #  ADD edx, [edi + 0x10]

3   ADD,f ecx, esi, ecx @ & flags      #  ADD ecx, esi

4   ST ecx, [edi + 0x8]                #  MOV [edi + 0x8], ecx

5   LD [edi + 0xc], r1                 #
6   CMP,f r1, esi, flags               #  CMP [edi + 0xc]

7   CCTOLP  flags|nc -> p[0]           #
8   CT Label, p[0]                     #  JNC Label

9   LD [edi + 0x14], r1                #
10  SUB,f edx, r1, edx & flags         #  SUB edx, [edi + 0x14]

Label:
11  ST edx, [ecx]                      #  MOV [ecx], edx

12  SRL,f esi, 1, esi & flags          #  SRL esi, 1
```

Please replace paragraph 2 beginning at line 4 on page 19 with the following amended paragraph:

After that real exception at operation "SUB" will occur. Such kind of exception ("diagnostic operand") is precise at the host platform and the whole third wide instruction won't be executed. Then host exception handler ~~932~~ 532 (Fig. 5) will be invoked. It takes wide instruction address from the RPR register and extracts correspondent documentation for the Recovery Point being addressed by the value of instruction pointer in the RPR register. On the base of this documentation the exception handler determines a foreign operation which the process of recovery process should be ~~stared~~ started from.

Replace paragraph 3 beginning at line 21 on page 20 with the following amended paragraph:

Exception handler 532 invokes a dynamic binary translator 534. Dynamic binary translator 534 may be the same as binary translator 203 or it may be a simplified version thereof. When exception handler 532 invokes binary translator 534, the foreign code 520 is accessed

beginning at the nearest recovery point, which in Figure ~~9~~ 5, corresponds to instruction 524. Using the run-time values, binary translator 534 regenerates sequential binary translated code as indicated at 536. The first instruction in code sequence 536 corresponds to the recovery point, that is, instruction 538. Execution proceeds to the instruction that would correspond to instruction 518, which caused exception 530. If the exception, such as exception ~~940~~ 540 re-occurs, exception handler 532 will report the problem to the foreign exception handler which, if not already exist in binary translated form, will be binary translated into the host code. Otherwise, execution proceeds to the next control transfer operation and switches back to the optimized binary translated code 510. It will be appreciated that any exception occurring between Recovery Points requires that exception handler 532 begin interpreting foreign code 520 from a correspondent Recovery Point in the foreign code. Accordingly, the foreign context in optimized binary translated code must not be changed irretrievably between Recovery Points. Thus, preliminary register information must be retained and the registers cannot be reused until execution reaches a recovery point. The advantage of this method is that the dynamic binary translator 534 eliminates the optimization by re-executing the code in a sequential manner. It affords the opportunity to easily reconstruct the conditions that caused the exception in the optimized code and restore the correct foreign context to the state just prior to the exception.